Table of Contents

1
Writing in ECMAScript 2015+

For the rest of this book, we will be using syntax and features introduced in ECMAScript 2015 (ES6) of JavaScript. This chapter provides a non-comprehensive run-through of the features we'll be using in the rest of the book. Specifically, we will cover:

- Declaring variables with `let` and `const`
- Setting defaults
- Destructuring assignment
- Rest and spread operators
- Template literals
- Promises

ES6 also added a native implementation of **modules**, but because explaining modules in JavaScript requires a lot of context and history, I've relegated the treatment of ES6 modules to `Chapter 4`, *Setting Up Development Tools*. Also, we have already discussed arrow functions in `Chapter 3`, *Important concepts in JavaScript*, so we won't revisit them here.

If you see any new ES6 (or newer) syntax that is not covered here, apart from searching on Google, you may also find the Babel REPL (`http://babeljs.io/repl/`) useful. Babel is a tool that translates ES6+ code into older versions of JavaScript. Babel's website provides an online **Read–Eval–Print Loop (REPL)** tool that evaluates any code we paste into it and prints out the equivalent ES5 code.

There are other tools similar to the Babel REPL, such as TypeScript playground (`http://www.typescriptlang.org/play/`), Traceur Transcoding Demo (`http://google.github.io/traceur-compiler/demo/repl.html`), and the Closure Compiler (`https://closure-compiler.appspot.com/home`), although some of these tools do a

little more than just simple compilation.

We will cover Babel in much more depth in `Chapter 4`, *Setting Up Development Tools*.

Declaring variables with let and const

Before ES6, the scope of an identifier depended on whether it was declared with the `var` keyword or not. `var` restricts the scope of the identifier to within the function it is declared in:

```
(function() {
  globalScoped = 1;
  var functionScoped = 1;
})();

globalScoped; // 0
functionScoped; // undefined
```

Here, inside the **immediately-invoked function expression (IIFE)**, the `functionScoped` identifier is declared within the function; this means any code inside the function's body, including the inner functions, is able to access the variable. Anything outside the function body cannot access, or even know about the existence of, the variable:

```
(function() {
  var functionScoped = 1;
  functionScoped; // 1
  (function() {
    // functionScoped is available from anywhere
    // within the function block
    // Even inside inner functions
    functionScoped; // 1
  })();
})();

functionScoped; // undefined
```

Then, we had `globalScoped = 1`. This is technically not a variable declaration; all that code is doing is trying to assign the value of `1` to the `globalScoped` identifier. But because `globalScoped` is not declared within the function, the JavaScript engine will go up one execution context level (into the global context) to try to find the next level at which `globalScoped` was declared. In our case, `globalScoped` was not

declared anywhere, and so a new property is added to the global object, `window` (in the case of a browser). This explains why, after the function that assigns a value to `globalScoped` has been invoked, we can access the `globalScoped` variable.

So before ES6, we only had one type of variable declaration: function-scoped.

Block-scoped, local variable declaration with let

While `var` is scoped inside the function block it is defined in, `let` is scoped inside *any block* it is defined in. In other words, `var` is **function-scoped**, `let` is **block-scoped**.

In JavaScript, a block is a grouping of zero or more statements that will be executed together, as if there were just one statement. Blocks are delimited by curly brackets, `{}`.

Note that object literals are also enclosed in curly brackets (`{}`), but your JavaScript engine will be able to decipher one from the other—if the curly brackets contain a list of statements, it'll be interpreted as a block.

Let's look at an example:

```
(function () {
  if (true) {
    var functionScoped = 0;
    let blockScoped = 0;
  }
  functionScoped; // 0
  blockScoped; // Uncaught ReferenceError: blockScoped is not defined
})()
```

The `functionScoped` variable is accessible from anywhere within the scope of the function body (it's even accessible before the variable has been declared, due to a feature called **hoisting**):

```
(function () {
  functionScoped; // undefined, but does not throw an error
  if (true) {
    var functionScoped = 0;
  }
  functionScoped; // 0
})()
```

`let`, however, was defined within the `if` block, and so is inaccessible from outside the block.

Declaring constants with const

`const` is the same as `let`, except `const` means the identifier cannot be reassigned to a different variable. This, however, does not mean the *value* of the variable cannot be changed—the value is not immutable—it is the binding between the identifier and the variable that is immutable. Let's look at a few examples:

```
const answer = 42;
answer = 50; // Uncaught TypeError: Assignment to constant variable.

const phone = { battery: 78 };
phone = { battery: 77 }; // Uncaught TypeError: Assignment to constant
variable.
```

Both of the examples here threw an error because we are reassigning the identifiers (`answer` and `phone` in our case) to a new variable; this breaks the binding between the identifier and the variable.

Note that in our `phone` example, even if we assign an object that's identical *in value* to the previous one, it'll still throw an error because objects are stored and retrieved *by reference*, not by value; to JavaScript, they are simply two different objects that happen to hold the same values:

```
const phone = { battery: 78 };
phone = { battery: 78 }; // Uncaught TypeError: Assignment to constant
variable.
```

However, mutating the value of the object itself is fine, because we are not reassigning a different reference to the identifier, we're just mutating *what* it references:

```
const phone = { battery: 78 }
phone.battery = 77; // OK
```

If you want a variable to be immutable, check out Immutable.js (`https://github.com/facebook/immutable-js`) by Facebook, or Mori (`https://github.com/swannodette/mori`); both can impose immutability on data.

Comparing let and const

With ES6, you never have to use `var`. So the question remains: when should you use `let` and when should you use `const`?

The answer is simple: use `const` by default, and only use `let` when you need to reassign something to the same identifier, such as in a loop.

But why favor `const` over `let`? Because other parts of the code might rely on the identifier to point to their expected value; if that value is reassigned, the code that relies on it might throw errors. In an extreme example, imagine if someone reassigned the `window` identifier to be an empty object:

```
window = {}; // Bad
```

That's why you should use `const` by default, so that each time you need to use `let`, you have to think about whether it is really necessary to introduce a variable that you want to mutate. For example, the following code requires the use of `let`:

```
function sumAndDouble(...numbers) {
  let total = 0;
  for (let i = 0; i < numbers.length; i++) {
    total += numbers[i];
  }
  total = total * 2;
  return total;
}
```

However, this one does not:

```
function sumAndDouble(...numbers) {
 const sum = numbers.reduce((total, number) => total + number, 0);
 const sumDoubled = sum * 2;
 return sumDoubled;
}
```

Mutating data, even the binding between identifiers and variables, can lead to errors. This is why `Immutable.js` (facebook.github.io/immutable-js/) and similar libraries have risen in popularity—to minimize errors caused by mutable data:

```
window.console = {};
console.log("Hello"); // Uncaught TypeError: console.log is not a
function
```

Providing defaults

Providing **defaults** for function parameters allows you to write more succinct code. Instead of this:

```
function addPoint(x, y) {
  x = typeof x === 'undefined' ? 0 : x;
  y = typeof y === 'undefined' ? 0 : y;
  drawPoint(x, y);
}
```

We can now write this:

```
function addPoint(x = 0, y = 0) {
  drawPoint(x, y);
}
```

Let's have another example:

```
function addRectangle(x = 5, y = 2*x) {
  console.log(x);
  console.log(y);
  drawRectangle(x, y);
}

addRectangle(); // 5, 10
addRectangle(9); // 9, 18
addRectangle(9, 12); // 9, 12
```

And here's what's happening:

- We are able to use `y = 2*x` as the last function parameter because parameters that are already defined (such as `x`) can be used in the default assignment of subsequent parameters
- If neither parameters were provided, `x` would default to `5`, and `y` would default to double the value of `x`, which is `10`
- If `x` was provided but `y` was not, `y` would still default to double the value of `x`, so in our example it would be `18`

That's the basics of defaults, but as you'll see in later sections of this chapter, you can combine using defaults with destructuring assignment to achieve some powerful shorthands.

Destructuring assignment

Destructuring assignment is a new syntax that allows you to unpack the components of an array or object into distinct variables with just one line:

```
// Arrays
[x, y] = [0, 1];
x; // 0
y; // 1

// Objects
({ color, margin } = { color: "red", margin: 20 })
color; // "red"
margin; // 20
```

We had to wrap our object literals with parentheses; otherwise, the interpreter would interpret them as a block and throw an `Uncaught SyntaxError: Unexpected token =` error.

We'll talk about array-destructuring first, and then object-destructuring.

Array-destructuring

In **array-destructuring**, the variable assignment would start from the left of the array, working its way to the right, until it runs out of values (in which case, `undefined` would be used):

```
const [a, b, c] = [1, 2]
a; // 1
b; // 2
c; // undefined
```

You can also combine array-destructuring with other new features of ES6, such as defaults and the rest operator:

```
// Using defaults
const [x = 9, y = 8] = [1]
x; // 1
y; // 8
```

Use cases for array-destructuring

Array-destructuring is often used to collect grouped matches from regular expressions (RegEx). For example, you might want to write a function that breaks down different portions of a URL, such as this scheme:

```
const URL_REGEX = /([a-zA-Z]*):\/\/(?:([a-zA-Z]*):([a-zA-Z]*)@)([A-Za-
z0-9]*)\.([a-zA-Z]*)(?::(\d{1,5}))?(?:\/([a-zA-Z0-9\/]*))(?:\?([a-zA-
Z0-9=&]*))?(?:#([a-zA-Z0-9-]*))?/;
const urlFragments =
URL_REGEX.exec("protocol://username:password@hostname.ext:30000/a/path
/?a=1&b=2#hash");
```

Here, `urlFragments` would be an array:

```
// urlFragments
[
"protocol://username:password@hostname.ext:30000/a/path/?a=1&b=2#hash"
,
  "protocol",
  "username",
  "password",
  "hostname",
  "ext",
  "30000",
  "a/path/",
  "a=1&b=2",
  "hash"
]
```

To assign a variable to each of the groups, we can painstakingly list them all out and assign each, one by one:

```
const url      = urlFragments[0];
const protocol = urlFragments[1];
const username = urlFragments[2];
const password = urlFragments[3];
const hostname = urlFragments[4];
const ext      = urlFragments[5];
const port     = urlFragments[6];
const path     = urlFragments[7];
const query    = urlFragments[8];
const hash     = urlFragments[9];
```

Or, we can use array-destructuring to extract the values from the array:

```
[url, protocol, username, password, hostname, ext, port, path, query,
hash] = urlFragments;
```

Object-destructuring

Object-destructuring acts in a similar manner to array-destructuring, but since object keys are not ordered, we must pass in the keys we want to extract from the object:

```
const { color, fontSize } = { color: "red", fontSize: 16 };
color; // "red"
fontSize: // 16
```

You can also reassign the key name to your own variable name using a colon (`:`):

```
const res = { points: 68 };
const { points: score } = res;

score; // 68
points; // Uncaught ReferenceError: points is not defined
```

You can extract values from nested objects and combine them with array-destructuring:

```
const user = {
  username: "d4nyll",
  settings: {
   public: true
  },
  emails: [{
    email: "dan@danyll.com",
    verified: true
  }, {
    email: "dan@brew.com.hk",
    verified: false
  }]
}

const {
  username, // Normal object destructuring
  settings: { public } // Extracting nested object
  emails: [{email:primaryEmail}, {email:secondaryEmail}], // Nested
object with array destructuring
} = user;

username; // "d4nyll"
public; // true
primaryEmail; // "dan@danyll.com"
secondaryEmail; // "dan@brew.com.hk"
```

Use cases for object-destructuring

Object-destructuring is often used to extract option values from the parameters of a function. Often, when an API's endpoints can accept many options (some of which may be optional), instead of having them written in a long list, you can pass in an `options` object instead.

For example, if an API method has the following syntax:

```
SomeAPI.method(requiredA, requiredB[, optionalC, [optionalD,
[optionalE, [optionalF]]]]);
```

Then, if we want to pass in the `requiredA`, `requiredB`, and `optionalF` options, we'd have to write this:

```
// Passing in arguments as a list
SomeAPI.method(requiredA, requiredB, undefined, undefined, undefined,
optionalF);
```

Instead, we can pass in an `options` object, which allows us to specify only the data we want:

```
// Passing in arguments as an object
SomeAPI.method({
  requiredA: requiredA,
  requiredB: requiredB,
  optionalF: optionalF
});

// Passing in arguments as an object, using property value shorthand
SomeAPI.method({ requiredA, requiredB, optionalF });
```

I'm sure you'll agree with me that for methods with many parameters, passing in a single object containing all the options is neater.

Now, where object-destructuring fits in is in the extraction of values from these parameter objects:

```
SomeAPI.addText = function (options) {
  { text, color, fontFamily, fontSize } = options;
}
```

We can even combine object-destructuring with function defaults to provide default values for the options:

```
addText = function (options) {
  const { text = "", color = "red", fontFamily = "'Lato', sans-serif",
```

```
  fontSize = 16 } = options;
    console.log(text);
    console.log(color);
    console.log(fontFamily);
    console.log(fontSize);
}

addText({
    text: "Hello World!",
    fontSize: 20
}); // "Hello World!", "red", "'Lato', sans-serif", 20
```

Or, we can even eliminate the use of the `options` variable:

```
addText = function ({ text = "", color = "red", fontFamily = "'Lato',
sans-serif", fontSize = 16 } = {}) {
    console.log(text);
    console.log(color);
    console.log(fontFamily);
    console.log(fontSize);
}
```

Understanding Rest and Spread

Defaults and destructuring-assignment allow you to keep your code more concise and readable; the **rest** and **spread** operators do the same.

At the time of writing, the rest and spread operators only work on arrays, but there is a stage-four proposal to expand them to objects, which you can follow at `github.com/tc39/proposal-object-rest-spread`. We will examine the use of these operators for both arrays and objects, but just remember that to use rest/spread on an object will require a polyfill, such as the `@babel/plugin-proposal-object-rest-spread` plugin.

Both operators use the `...` syntax, so it can be confusing at the beginning to remember which is which; at the end of this chapter, I'll give you some tips on how to remember their names. But first, let's look at the spread operator.

Combining Iterables with the Spread Operator

The spread operator *spreads* its elements out into its containing iterable:

```
const teachers = ["ann", "bob", "carol"];
```

```
const support = ["yasmin", "zach"];
const staff = [...teachers, "sam", ...support]; // "sam" is the
receptionist
staff; // ["ann", "bob", "sam", "carol", "yasmin", "zach"]
```

The elements of the `teachers` and `admins` arrays have been copied into the `staff` array, at the position where they were specified.

This is a common use case for spread operators: to combine/concatenate arrays in a concise manner. To express this logic without spread operators, we'd have to do something like this:

```
const staff = [].concat(teachers, ["sam"], support);
```

This is more bloated and harder to read. You can do the same for objects as well (with a polyfill). Here's an example that uses spread operators to style a button:

```
const buttonBase = {
  borderColor: "#ff6600",
  borderStyle: "solid",
  borderWidth: "1px",
}
const roundButton = {
  ...buttonBase,
  borderRadius: "50%"
}

const myButton = {
  ..roundButton,
  borderColor: "#8711e4"
}
```

The `roundButton` object will contain all the properties of the `buttonBase` object, plus the additional `borderRadius` property; likewise, `myButton` will contain all the properties of `roundButton`, plus the `borderColor` property, which will override the `borderColor` property defined in the `buttonBase` object.

This pattern means styles don't have to be duplicated and allows you to compose new styles by combining multiple base styles.

Collecting variables with the rest operator

While the spread operator spreads its elements out, the rest operator collects multiple elements and "condenses" them into one variable. Let's start with an example:

```
// Using rest operator to collect the rest of the array
const [a, b, ...c] = [1, 2, 3, 4, 5, 6];
a; // 1
b; // 2
c; // [3, 4, 5, 6]
```

This is commonly used in function parameters where the function can act on an indefinite number of parameters. Without the rest operator, you'll have to iterate through the `arguments` object:

```
function add() {
  let total = 0;
  for (let i = 0; i < arguments.length; i++) {
    total = total + arguments[i];
  }
  return total;
}
add(1, 2, 3); // 6
```

It might look like the arguments object is already an array, and we can just use the `.reduce` function on it, but in fact, it's just a special *array-like object* with `length` (and other) properties, and where the rest of the keys are integers.

Using `console.log` on the `arguments` object would give you something like this: `{"0":1,"1":2,"2":3}`.

With the rest operator, however, it can become much cleaner and more concise:

```
function add(...numbers) {
  return numbers.reduce((total, num) => total + num, 0);
}

add(1, 2, 3); // 6
```

Destructuring rest parameters

You can combine the rest operator with destructuring. There are many use cases for this; we will outline just one here.

Let's say we have a publishing/blogging platform, and we have exposed a method that allows users to change the content of their posts. We want to give them a lot of freedom, so they can change anything they want, but we want to make sure they don't change the `id` and `createdAt` fields. We can implement something such as this:

```
const updatePost = function (newPost) {
  const { id, createdAt, ...sanitizedPost } = newPost;
  console.log(sanitizedPost);
  collection.update(id, sanitizedPost);
}

const newPost = {
  id: "521",
  createdAt: 1498160284666,
  title: "New title",
  description: "New description"
};

updatePost(newPost); // {title: "New title", description: "New
description"}
```

We used destructuring assignment to pull out the `id` and `createdAt` properties, collected the rest of the properties with the rest operator, and placed it in the `sanitizedPost` variable.

Performing string interpolation with template literals

Template literals, also known as **template strings** in earlier versions of ES6, allow you to interpret variables inside a string while keeping it easy to read.

The syntax of the template literal encloses a string inside backticks (`` ` ``, or the grave accent), as opposed to single (`'`) or double quotes (`"`). Then, inside the template literal, you can define placeholders with `${}`, and any expressions between the curly brackets are interpreted as JavaScript; this is known as **string interpolation**:

```
var name = "Daniel";
var getLocation = function() { return "Hong Kong"; };

// equivalent to "My name is Daniel and I am from Hong Kong!"
`My name is ${name} and I am from ${getLocation()}!`
```

You can only put **expressions** inside the curly brackets; **statements** are not allowed.

In JavaScript, expressions evaluate to a value. For example, `1 + 2` is an expression, as it produces the value of `3`; a function call is also an expression, as the function will always return a value, even if it's an implicit `undefined`.

Statements perform an action. For example, control flow constructs, such as loops or conditions, are examples of statements.

Some common statements have an expression equivalent. For example, a simple `if/else` can be expressed as a ternary operator.

Before template literals were introduced, you had to painstakingly concatenate the strings together:

```
var name = "Daniel";
var origin = "Hong Kong"

// ES5 with concatenation
'My name is ' + name + ' and I am from ' + origin + '!'

// ES6 with template literal
`My name is ${name} and I am from ${origin}!`
```

Although it is subjective, most would agree that the template literal format is more readable.

While string interpolation is new to JavaScript, it is already a long-established feature for other programming languages, such as Python and C#:

```
# Python
'My name is %s and I am from %s!' % ('Daniel', 'Hong Kong')

/// C#.NET
string.Format("My name is {0} and I am from {1}!", "Daniel", "Hong
Kong");
```

Improving readability with multiline strings

Another aspect in the readability of template literals is multiline strings. Instead of having to write `\n` each time you want a new line, you can simply create a new line:

```
// ES5 with `\n`
var bio = "I like dogs.\nI also like cats.\nBut not so much spiders!";

// ES6 with template literals
var bio = `I like dogs.
I also like cats.
But not so much spiders!`;
```

Tagging template literals

The last feature of template literals is a little more complicated, and requires passing the template literal into a function for it to be evaluated. It'll be clearer with an example:

```
function translate(a, b, c, d) {
 console.log(a); // Array of ["My name is ", " and I am from ", "!"]
 console.log(b); // "Daniel"
 console.log(c); // "Hong Kong"
 console.log(d); // undefined
}

var name = "Daniel";
var origin = "Hong Kong";

// "My name is Daniel and I am from Hong Kong!"
translate`My name is ${name} and I am from ${origin}!`;
```

As you can see here, we tagged our template literal with the `translate` function. This passes, as the first argument, an array of strings that compose the template literal, splitting the strings where the placeholders are:

```
`My name is ${name} and I am from ${origin}!`

[ "My name is ", " and I am from ", "!"]
```

Then, subsequent arguments are passed the values of the placeholders, in the order they appear in the template literal.

While this sounds strange and esoteric, this is actually how template literals work in the first place, just that instead of specifying a function, a default function is used—one that simply concatenates the strings in order.

Tagged template literals are commonly used in some specific areas. In the remainder of this section, we will highlight one use case: translations.

Tagged template literals are also used in frontend development to create **styled components**. Read more about them at `github.com/styled-components/styled-components`.

Translations

Many platforms are internationalized, meaning they cater to audiences from different locales who understand different languages. Part of the **internationalization** (**i18n**) process is **localization** (**l10n**), or **translation**.

There are many libraries and services that help with translation, such as `gettext` and `Weblate`. Most of them separate translations into files, one for each language, and store the strings in a key-value format:

```
// en-gb.json
{
  "hello": "Hello"
}

// zh-hk.json
{
  "hello": "你好"
}
```

This works fine for single words or phrases, but what if you want to store complete sentences with placeholders for variables? You'd have to break down your sentence and store each part individually:

```
Your delivery will arrive between X and Y!
```

This sentence might be implemented like so (where we use `t()` as a library-agnostic function to translate a string):

```
// en-gb.json
{
  "deliveryMessage1": "Your delivery will arrive between ",
  "deliveryMessage2": " and ",
  "deliveryMessage3": "!"
}

// main.js
var start = "X";
var end = "Y";
var message = t("deliveryMessage1") + start + t("deliveryMessage2") +
end + t("deliveryMessage3");
```

However, with template strings, you can store the complete sentence as a single string and use `eval()` to evaluate the string as code, converting our string into a template literal:

```
// en-gb.json
{
  "deliveryMessage": "Your delivery will arrive between ${start} and
${end}!"
}

// main.js
var start = "X";
var end = "Y";
var template = t("deliveryMessage");
var message = eval("`" + template + "`");
```

However, using `eval()` might not be the best option, because it runs whatever code it is passed, so a malicious user can inject malicious code into our program. I hope this goes without saying, but that's very bad for security. Furthermore, you cannot debug code that is executed inside the `eval` call. Instead, we can do something like this:

```
// en-gb.json
{
 "Your delivery will arrive between ${} and ${}!": "Your delivery will arrive between ${}
 and ${}!"
}
// zh-hk.json
{
 "Your delivery will arrive between ${} and ${}!": "您的貨品將於${}和${}之間到達！"
}

// main.js
var translate = function (template, ...values) {
  // Recreate the key
  var key = template.join("${}"); // "Your delivery will arrive between ${} and ${}!"

  // Get the translated template
  var translatedTemplate = t(key); // "您的貨品將於${}和${}之間到達！"

  // Replace the placeholders with the actual values
  var translatedTemplateFragments = translatedTemplate.split("${}");
  var translatedString = translatedTemplateFragments.map(function(fragment, index) {
    return fragment + (values[index] || "");
  }).join("");

  return translatedString;

}
var start = "X";
var end = "Y";
var message = translate`Your delivery will arrive between ${start} and ${end}!`
```

Although it requires a bit more logic to implement the `translate` function, it has several benefits:

- The structure of the entire sentence is stored in the language file, giving the translator more context
- Better readability using the tagged template literal

- Deals with languages that have different sentence structures—more specifically, differences in how they order components of a sentence. For example, our string, "Your delivery will arrive between X and Y!", might be translated to "您的貨品將於Y和Z之間到達！" in Chinese.

Literally translating the Chinese text without rearranging the order of the phrases, we get "Your products will at Y and Z between arrive":

```
Your (您的) products (貨品) will (將) at (於) Y and (和) Z between (之間) arrive (到達)!
```

So, you might end up with two translation files that look like this:

```
// en-gb.json
{
  "deliveryMessage1": "Your delivery will arrive between ",
  "deliveryMessage2": " and ",
  "deliveryMessage3": "!"
}
// zh-hk.json
{
 "deliveryMessage1": "Your products will at ",
 "deliveryMessage2": " and ",
 "deliveryMessage3": " between arrive!"
}
```

Obviously, `"Your products will at"` shouldn't be translated using the same key as `"Your delivery will arrive between"`, but that's what you'll have to do with string concatenation.

Running asynchronous operations with promises

Lastly, we will look at **promises**. Promises are a long-established pattern in the JavaScript ecosystem, long before it was standardized as part of ECMAScript. Bluebird, (`https://github.com/petkaantonov/bluebird`) arguably the most popular library to implement promises, was first released in 2013. We'll begin this section by discussing the problems promises are trying to solve, the way promises solve them, and how promises actually work. So, let's start with some code:

```
const getProfile = function (id) {
  const profile = ExternalAPI.retrieveProfileSync(id);
  return profile;
```

```
}
const profile = getProfile(324);
displayProfile(profile);
```

This code is written in a synchronous style, where an operation is only executed after the previous one has completed; in other words, unless the preceding statement has completed, any downstream statements cannot be executed.

JavaScript is single-threaded, which means only one operation can occur at any one time. So, when the engine is waiting for the external API to return with the results, nothing else can be executed. This means synchronous code is **blocking**.

Eventual completion using callbacks

The way JavaScript deals with this is through introducing asynchronous features. The most primitive one is a **callback**, which is simply a function that'll be called after a certain function has occurred. We can change the previous `getProfile` function to execute asynchronously by incorporating a callback into it:

```
const getProfile = function () {
  ExternalAPI.retrieveProfile(324, function(error, profile) {
    if(error) { throw new Error(error.message);}
    displayProfile(profile);
  });
}
```

Instead of returning the `profile` object and then running a function to display the profile, you pass a callback into the `retrieveProfile` function, which will be called as soon as the server has returned with the results. This means you can still define logic that depends on other pieces of information, but without blocking the execution of the rest of the code.

Callbacks are very common in JavaScript; you've probably used callbacks when defining event listeners:

```
const button = document.getElementById("cta");
button.addEventListener('click', function() {
  // ...
});
```

Here, we passed a callback function into the `addEventListener` method.

Layered callbacks cause callback hell

We only had one callback in our previous example, but it's not uncommon to have many layers of callbacks. In the following example, we demonstrate a three-layered set of callbacks:

```
const getFeed = function () {
  ExternalAPI.getFollowing(function(error, followings) {
    if (error) { throw new Error(error.message) }
    const followingIds = followings.map(following => following.id);
    ExternalAPI.getPostsOfUsers(followingIds, function(error, posts) {
      if (error) { throw new Error(error.message) }
      const postIds = posts.map(post => post.id);
      ExternalAPI.getComments(postIds, function(error, comments) {
        if (error) { throw new Error(error.message) }
        displayProfile();
      });
    });
  });
};
```

In the example, we first call an endpoint to retrieve a list of users our user is following; then, we get the posts from those users; lastly, we get all the comments from those posts. This code is very hard to read, for the following reasons:

- There are many levels of indentation, making it difficult to elucidate, at first glance, which level a particular line of code belongs to
- Because of the indentation, the length of each line can get very long (the longest line here is 71 characters, with a two-space indentation style)
- It's hard to keep track of all the open/close brackets
- There is a lot of repetition regarding how the errors are handled

This is such a common problem that developers have termed it **callback hell**. There are several ways to mitigate this; one way is to separate each of these into different, named functions:

```
function getFeed () {
  ExternalAPI.getFollowing(function(error, followings) {
    if(error) { throw new Error(error.message);}
    const followingIds = followings.map(following => following.id);
    getPosts(followingIds);
  });
};

function getPosts (userIds) {
  ExternalAPI.getPostsOfUsers(userIds, function(error, posts) {
```

```
    if(error) { throw new Error(error.message);}
    const postIds = posts.map(post => post.id);
    getComments(postIds);
  });
};

function getComments (postIds) {
  ExternalAPI.getComments(postIds, function(error, comments) {
    if(error) { throw new Error(error.message);}
    displayProfile();
  });
};
```

While this code solves the issue of having too many levels of indentation, there's still a lot of repetition. Furthermore, someone reading the code has to jump from one place to the next in order to follow the logic. So, while it might be an improvement, it's certainly not ideal.

Avoiding callback hell with promises

Instead, we can implement the same function using promises, avoiding callback hell:

```
const getFeed = function () {
  WrapperAPI.getFollowing()
    .then((users) => users.map(user => user.id))
    .then(WrapperAPI.getPostsOfUsers)
    .then((posts) => posts.map(post => post.id))
    .then(WrapperAPI.getComments)
    .then(comments => displayProfile())
    .catch(error => throw new Error(error.message))
}
```

We'll go into how promises work shortly, but you can see that the code is already much more readable. Furthermore, we've stopped repeating our error-handling function and have defined it only once at the end.

Mechanism of promises

So how does a promise work? Let's start at the top:

```
const getFeed = function () {
  WrapperAPI.getFollowing()
   .then((followings) => {
     return followings.map(following =>following.id);
```

```
    })
    ...
```

First, `ExternalAPI.getFollowing` uses a callback, but we want the method to return a `Promise`; therefore, we must first wrap the `ExternalAPI` methods using the `Promise` object's constructor:

```
WrapperAPI.getFollowing() {
  return new Promise(function (resolve, reject) {
    ExternalAPI.getFollowing(function(error, followings) {
      if(error) { reject(error); }
      const followingIds = followings.map(following => following.id);
      resolve(followingIds);
    });
  });
}
```

In our wrapper function, we wrapped the `ExternalAPI.getFollowing` method, which uses a callback, and returned a `Promise` object. We can repeat this wrapping for any function that uses a callback and doesn't support promises.

But what is a promise? What are the `resolve` and `reject` parameters?

What is a promise?

The best description for the `Promise` object comes from the Promises page on Mozilla Developer Network (`developer.mozilla.org/en/docs/Web/JavaScript/Reference/Global_Objects/Promise`), which states:

> *"The Promise object represents the eventual completion (or failure) of an asynchronous operation, and its resulting value."*

With a callback, nothing is returned, which means you must nest callbacks or break each callback into its own function; as we have demonstrated, this is bad for readability (and thus maintainability).

But what can you return? We can't return the value of the operation, because we don't know what it is! But, we know that the asynchronous operation will *eventually* return, either successfully or with an error. So, we can return an object that represents this future result, and that object is the `Promise`.

You may think of a `Promise` as being a *promise* that a result will be provided in the future.

Structure of a promise

Let's look inside a `Promise` object and see its properties:

```
{
  [[PromiseStatus]]: "pending",
  [[PromiseValue]]: undefined,
  __proto__: { // inherits from Promise.prototype
    catch: function () { ... },
    constructor: function () { ... },
    then: function () { ... },
    Symbol(Symbol.toStringTag): "Promise"
  }
}
```

First of all, we can see that a `Promise` has its own internal properties: `[[PromiseValue]]` and `[[PromiseStatus]]`. `[[PromiseStatus]]` represents the status of the asynchronous operation, and can be in one of three states:

- `pending`: The asynchronous operation has been dispatched, but no results have been returned
- `fulfilled`: The operation has returned successfully
- `rejected`: The operation has returned with an error

When a promise is fulfilled or rejected, it is **settled** and will remain in that state permanently, meaning we cannot subsequently resolve a rejected promise, or reject a resolved promise. Some may also refer to this settled state as a **resolved** state.

`[[PromiseValue]]` is the value that'll eventually be returned. If the operation is successful, it'll be the result that's been returned; if the operation resulted in an error, it'll contain the `Error` object that was returned.

The executor function

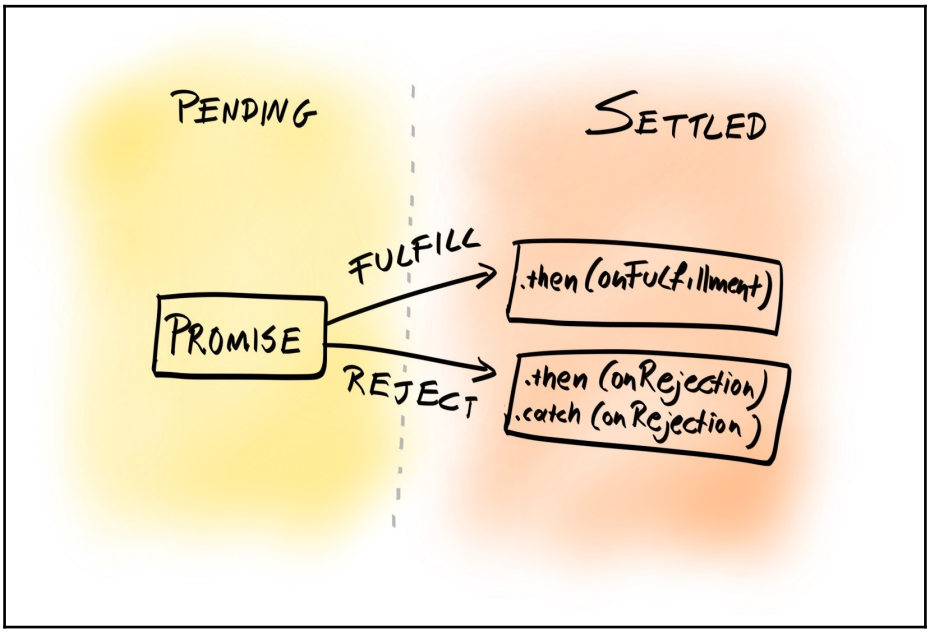
When we create a `Promise`, we must pass into it a single function accepting two arguments: `resolve` and `reject`. This function is known as the **executor function**, and is executed at the point when the promise is created. The JavaScript engine passes

in two native functions, internal to the engine, as arguments for the `resolve` and `reject` parameters.

When either of these internal functions is called, it will find the `Promise` object that it is linked to, and update its status and value; that is, its `[[PromiseStatus]]` and `[[PromiseValue]]` properties, respectively. After these values have been updated, it'll then call the `then` or `catch` methods of the same `Promise` object.

The `then` and `catch` methods are in the `__proto__` property of the `Promise` instance, which is inherited from the `Promise.prototype` object.

The following diagram illustrates the flow of how a promise moves from a pending state to a settled state:



The `then` and `catch` methods accept a function, which is the handler function to call if the results are returned successfully or unsuccessfully, respectively. This handler function is itself automatically wrapped inside another function that guarantees the return of a `Promise`.

Confused? Here's an example implementation of the `then` method:

```
const then = function (onResolved) {
  const results = onResolved();

  // If results is a promise, then simply return it
  if(results && typeof results.then === "function") {
    return results;
  }

  // Otherwise, wrap the result in a promise and return it
  else {
    return new Promise(function(resolve) {
      resolve(results);
    })
  }
}
```

This code shows that regardless of whether the expression inside the `then` method returned is synchronous and returns a value, or it returns a promise, the `then` method will always ensure a promise is returned. This allows promises to be chained one after the other:

```
const getFeed = function () {
 WrapperAPI.getFollowing()
 .then((users) => users.map(user => user.id))
 .then(WrapperAPI.getPostsOfUsers)
 .then((posts) => posts.map(post => post.id))
 .then(WrapperAPI.getComments)
 .then(comments => displayProfile())
 .catch(error => throw new Error(error.message))
}
```

The chaining mechanism might be more clearly explained when expressed with some temporary variables:

```
const getFeed = function () {
  try {
    const promise1 = WrapperAPI.getFollowing();
    const promise2 = promise1.then((users) => users.map(u => u.id));
    const promise3 = promise2.then((ids) =>
WrapperAPI.getPostsOfUsers(ids));
    const promise4 = promise3.then((posts) => posts.map(post =>
post.id));
    const promise5 = promise4.then((postIds) =>
WrapperAPI.getComments(postIds));
    const promise6 = promise5.then(comments => displayProfile());
  } catch (error) {
    throw new Error(error.message));
  }
}
```

A benefit of using the promise chain syntax rather than storing each promise in a temporary variable is that you can specify a single exception-handler for all previous operations, whereas with the latter you must wrap it inside a `try/catch` block, making it less readable.

With the promise chain, whenever an exception is thrown, the program will look down the promise chain, identify the first `catch` block, and call its handler. This is known as **error propagation**.

Generators, observables, thunks, and asynchronous functions

Using promises is a way to make asynchronous statements much more readable. There are more advanced features of promises that we have not covered, such as `Promise.all` and `Promise.race`, which I encourage you to study for your own pleasure.

Promises are not the only alternative to callbacks; **asynchronous functions** (or `async`/`await`) were introduced in ES8, and have been available since Node 7.6.0, Chrome 55, Firefox 52, Safari 10.1, and all versions of Edge. There are also other constructs that can work with, and alongside, promises, such as **generators**, **observables**, and **thunks**. Again, they are fascinating topics and you should study those patterns, so you'll be able to pick the best pattern for the situation.

Summary

In this chapter, we went through some of the most common ES6 syntax used today. We chose ES6 because it was the most significant version and introduces the most new syntax. However, ECMA plans to introduce a new version every year, and so I encourage you to keep a close eye on new proposals and releases at `https://github.com/tc39/ecma262`.

It's easy to start writing in ES6, since ES5 syntax is still valid in ES6, so you can convert your source code incrementally without breaking anything. You may want to try a tool called Lebab (`https://lebab.io/`), which allows you to transpile your ES5 code into ES6 (or even ES7/8) syntax.

In the next chapter, we will learn how to set up and keep a history of our application using a **Version Control System** (**VCS**) called Git.

Index